Scope and "In Scope"

- The **scope** of a variable is the range of a program's instructions where the variable is known
- a variable is **"in-scope"** if it can be accessed by its name in a particular part of a program.

Scoping strategies describe when variables are in-scope!

Lexical Environments

```
string dinner = "burgers";    // dinner has global scope; in scope everywhere

void party(int drinks) {      // drinks in scope for entire party() function
  cout << "Partay! w00t";
  if (drinks > 2) {
    bool puke = true;         // puke only in scope for this if statement
    cout << "Puked " << dinner;
  }
}

void study(int hrs) {         // hrs is in scope for the entire study() function;
different from in main()
  int drinks = 2;             // drinks is also in scope for study(); different from
in party()
  cout << "Study for " << hrs;
  party(drinks+1);
}

int main() {
  int hrs = 10;               // this hrs is only in scope in main(); different from
the hrs in study()!
  study(hrs-1);
}
```

- Functions have their own scope too: they are defined immediately after they're declared.
- **lexical environment** - set of in-scope variables and functions at a particular point in a program
  - in the `main()` function, the lexical environment only contains the variable `hrs` and the functions `main()`, `study()`, and `party()`
  - but, once we call `study(hrs-1)` and enter a new environment, the lexical environment changes: we get a different `hrs` (the one in the argument), and a new variable `drinks`

Languages have different scoping strategies:

- how do constructs like functions, control flow (if statements, loops), or classes affect variable scope?
- what do you do if you try to define a new variable with a name that already exists?
- when does a variable's scope end?

Lifetime

- a variable's **lifetime (extent)** describes when it can be accessed
    - includes times when the variable is in scope, but also times where it's not in scope (but can be accessed indirectly).
- *Values* also have lifetimes (that differ from variables)
    - a variable can go out of scope, but the underlying value it's bound to might not (eg: if another variable refers to it)

```
void study(int how_long) {
  while (how_long-- > 0)
    cout << "Study!\n";
  cout << "Partay!\n";
}

int main() {
  int hrs = 10;
  study(hrs);
  cout << "I studied " << hrs <<
        " hours!";
}
```

The scope of `hrs` is only within `main()`; when we call into `study`, we can no longer access it. But, *its lifetime also extends to* `study()`, since we can access it indirectly - through the variable `how_long`!

Python lets you manually end a variable's lifetime early:

```
def main():
  var = "I exist"
    ...
  del var    # no longer exists!
  print(var) # error!
```

- deletes the binding, not the underlying value

Lexical (Static) Scoping

- programs comprised of a series of nested **contexts**: files, classes in those files, functions in those classes, blocks in those functions, blocks within blocks, etc.
- **lexical scoping** - determine all variables that are in scope at a position X in our code by looking at X’s context first, then looking in successively larger enclosing contexts around x.

Languages like Python use the **LEGB** rule. We look for a variable in this order:

1. First, look at the **local scope**. In Python, this is either an expression or a function.
2. If it's not there, look at successive **enclosing scopes**. Repeat this until...
3. You hit the **global scope**.
4. If it's still not there, look at the **built-in scope** - this contains things like `print`

Importantly, once a context is over, the variables in that context can't be accessed!

Context Types

expression

```
sum([x*x for x in range(10)])     # ok
sum([x*x for x in range(10)] + x) # x isn't in scope!
```

block (in C++ roughly: any set of braces):

```
if (drinks > 2) {
  int puke = 5;
  // ...
}
cout << puke; // error - puke is not in scope!
```

Python doesn't scope wrt blocks!

```
def a():
  if True:
    x = 5
  print(x) # this works!
```

functions:

```
def a():
  x = 5

print(x) # error - x is out of scope!
```

Other relevant scoping constructs include:

- **classes/structs**: think about private functions and member variables!
- **namespaces**: think about `using namespace std;` versus having to type out `std::cout`
- the **global** scope: the last resort!

Shadowing

- **shadowing** - in an inner context, the redefined variable "replaces" all uses of the outer context variable
- once the inner context is finished, we return to the outer context variable.

```
// C++
int main(){
  int x = 42;
  for (int i = 0; i < 10; i++) {
    int x = i;
    std::cout << "x: " << x << '\n'; // prints values of i from 0 to 9
  }
  std::cout << "x:   " << x   << '\n'; // prints out 42
}
```

- Shadowing is not *great* practice (though there are legitimate uses!).
- However, you'll encounter it frequently when reading code, and *accidentally*!

```
def a(input):  # shadows the global input function
  print(input)
  a = input()  # doesn't work anymore!
```

Dynamic Scoping

- **Dynamic scoping** - the value of a variable is always the most recently defined (or redefined) version, regardless of the lexical scope!
  - These days, few languages use it
- what matters is the **chronological order** variables are defined in: not how the code is structured.

```
func foo() {
  x = x + 1;
  print x
}

func bar() {
  x = 10;
  foo();
}

func main() {
  x = 1;
  foo();
  bar();
}

// prints:
// 2
// 11
```

Interestingly, `foo()` or `bar()` didn't need arguments or parameters. After defining `x` once, its value "persists" across `foo()`, `bar()`, and the subsequent call to `foo()`! Neat!

Lisp example

```
(setq a 100)  # sets a to 100

(defun print_value_of_a ()
  (print a))  ; a is visible

; define local variable a, call print_value_of_a
(let ((a -42))
  (print_value_of_a))
```

Memory Safety (and Management)

- memory-safe languages prevent memory operations that could lead to *undefined* behaviors
- key property of strongly-typed languages

Memory unsafe languages allow:

- out-of-bound array indexes and unconstrained pointer arithmetic
- casting values to incompatible types
- use of uninitialized variables and pointers
- use of dangling pointers to dead objects (use-after-free, double-free, ...)

These are the *biggest* source of security vulnerabilities in low-level code.

```
int arr[10], *ptr = arr;
arr[-1] = 42;        // out-of-bound
cout << (ptr + 100); // pointer arith'c

int v;
Student *s = dynamic_cast<Student *>(&v);  // bad cast
s->study();

int val, *ptr;     // both uninitialized
cout << val;       // could leak info!
*ptr = -10;        // corrupts memory

Student *s = new Student("Gerome");
delete s;
s->study(); // dangling pointer
```

Memory safe languages

- throw exceptions for out of bounds array indexes; disallow pointer arith
- Throw an exception or generate a compiler error for invalid casts
- Throw an exception or generate a compiler error if an uninitialized variable/pointer is used;
- Hide explicit pointers altogether (e.g., Python)
- Prohibit programmer-controlled object destruction
- Ensure objects are only destroyed when *all* references to them disappear (Garbage Collection)

Memory Leaks

- should a language be considered unsafe if it can have memory leaks?
- No! they don't lead to undefined behaviour (predictably crashe if run out of memory)

Dealing with Memory Leaks

- **garbage collection** - The language manages all memory de-allocation automatically

- **Ownership model** - The compiler ensures objects get destroyed when their lifetime ends

    - Rust borrow-checker, C++ smart pointers

Garbage Collection

- automatic reclamation of memory which was allocated by a program, but is no longer referenced in code

- rule of thumb: garbage collect an object when there are no longer any references to that object

- no locals, member variables, globals

Advantages:

- eliminates memory leaks
- eliminates dangling pointers and the use of dead objects (prevents access to objects after they have been de-allocated)
- eliminates double-free bugs (inadvertent attempts to free memory more than once)
- eliminates manual memory management--simplifies code

Mark and Sweep

Mark

- discover all *active objects*, active objects must satisfy either:
  - *Root object* - global variables, local variables across all stack frames, and parameters on the call stack.
  - it is reachable from a root object. If an object can be transitively reached via one or more pointers/references from a root object (e.g.,robot object points to battery).

At a high-level, the GC identifies all root objects and adds their object references to a queue. Then, the GC uses the queue to BFS from the root objects and mark all the reachable objects as "in-use"

- each object is augmented with a bit which is set by the GC to mark that it is in use
- then dispose all unmarked objects

Here some pseudocode for a possible mark algorithm:

```
# Pseudocode for the Mark algorithm
def mark():
  roots = get_all_root_objs()
  candidates = new Queue()
  for each obj_ref in roots:
    candidates.enqueue(obj_ref)

  while not candidates.empty():
    c = candidates.dequeue()
    for r in get_obj_refs_in_object(c):
      if not is_marked(r):
        mark_as_in_use(r)
        candidates.enqueue(r)
```

Sweep

- traverse all memory blocks in the heap (each block holds a single object/value/array) and free all not in-use objects
- all memory blocks on heap are linked in a linked-list
- coalesce adjacent freed blocks -- reduce fragmentation

Here some pseudocode for the sweep algorithm.

```
# Pseudocode for the Sweep algorithm
def sweep():
    p = pointer_to_first_block_in_heap()
    end = end_of_heap()
    while p < end:
        if is_object_in_block_in_use(p):
            reset_in_use(p)      # remove the mark, object lives
        else:
            free(p)                        # free this block/object
        p = p.next
```

Pros:

- relatively simple
- no trouble with cyclic references (more on this later!)

Cons:

- program must be paused during GC, causing "freezes" (this is often called a "stop-the-world" GC)
    - bad for real-time programs
- dealing with large amounts of data can lead to thrashing (at the cache/page-level)
- causes memory fragmentation -- we'll have chunks of empty memory
- unpredictable!

Memory Fragmentation

- heap becomes peppered with small, unused memory blocks
- becomes slow (or impossible) to find space for new allocations

Mark and Compact

- mark as before
- instead of sweeping, compact all marked/in-use objects to a new contiguous memory block
- then adjust the pointers to the proper relocated addresses
- original block of memory is treated as if it's empty, can be reused without doing sweeping

The pros and cons are pretty similar to mark and sweep. In comparison, the only differences are:

- it much better deals with fragmentation
- but more complex to implement, requires more RAM, and is slower!

Interlude: Unpredictability

- only GC when there's **memory pressure** (when the program needs memory).
- GC is non-deterministic: you can't necessarily predict when (or even if) GC will run
- random slows in performance
- can't rely on destructors/finalizers to do things like release sockets, files, or other resources - **since you don't know when GC will run**!

> *GC not really used in real-time devices that need predictable behavior, use langauges w/o GC like C, Rust*

Reference Counting

- each object has a hidden counter that tracks how many references there are to it
- Every time a reference is created or destroyed, simply change the counter

References can *only* change:

- with assignment (`=`)
- when an object goes out of scope

Two hidden catches:

1. cyclic references are a *huge* problem; since there's a cycle, the counter will never reach zero

- languages that use RC need to explicitly deal with cycles in a different way.

2. a "cascade" can happen: if one object is destroyed, it could remove the last reference to another object, which then gets destroyed; then, that removes the last reference to another object, ... which could cause a large chunk of deletion (and could slow your computer down)!

- some languages amortize deletion over time -- but this has similar issues with non-determinism as tracing algorithms!

Pros:

- simple
- usually real-time (since reclamation is *usually* instant)
- more efficient usage since blocks are freed immediately

Cons:

- updating counts needs to be thread-safe (this is a **huge** issue!)
- updating on *every* operation could be expensive (both in time and space)
- cascading deletions
- requires explicit cycle handling
- fragmentation

Destructors

- used in manual memory management
- deterministic rules that govern when destructors are run, so the programmer can ensure *all* of them will run, and control *when* they run
- therefore can use destructors to release critical resources at the right times (freeing other objects, closing network connections, deleting files)

Finalizers

- in GC languages, memory is reclaimed automatically by the garbage collector
- finalizers are used to release unmanaged resources like file handles or network connections, which aren't garbage collected
- finalizer may not run at a predictable time or at all
- considered a last-line of defense for freeing resources, and often not used at all

Disposal methods

- a disposal method is a function that the programmer must manually call to free non-memory resources (e.g., network connections)

- use in GC languages because you can't count on a finalizer to run
- disposal provides a guaranteed way to release unmanaged resources when needed. However, we run the risk of forgetting to call the disposal method.

Mutability/Immutability

Class Immutability

- *all* objects of a class are immutable after construction

Object Immutability

- some objects of a particular class immutable

```
int main() {
  Nerd j("Joe",200); // mutable!
  const Nerd n("Carey",100);
  n.setIQ(120); // ERROR!
}
```

Assignability Immutability

- variable may not be re-assigned, but mutations can be made to original referred-to object

```
public static void someFunc() {
  final Nerd n = Nerd("Carey",100);
  n.setIQ(120);        // OK!!!
  n = Nerd("Joe",200); // ERROR!
}
```

Reference Immutability

- can prevent mutable object from being mutated via reference that's marked as immutable

```
void examine(const Nerd& n) {...}
```

Benefits

- eliminate aliasing bugs
- reduce multi-threading bugs (immutability avoids race conditions)
- eliminate identity variability bugs

```
map[x] = y;
x.change_identity();
cout << map[x]; // ??
```

- eliminate **temporal coupling bugs** - programmer does some initialization out of order (or not at all) resulting in incomplete object

```
Circle c = new Circle();
c.setRadius(10);
c.getArea();  // ??
```

- no hidden side effects

- easier testing--fewer failure modes
- runtime optimizations
- easy caching